IZ*Net

Nicholas Pun

Abstract. We develop the definitive method to differentiate between the JoYuriz trio.

Contents

1 Introduction

For the uninitiated, IZ*One[1] is a Korean-Japanese pop girl group. At the time of writing, their popularity allowed for a large volume of training data to be scraped from the internet and hence have become the focus of the network(s) designed. The name of this project "*IZ*Net*" is derived from their name and the abstract is a joke on that there are 3 members that the audience often have trouble differentiating.

Now, to get into talking about the project. We begin by completely contradicting the abstract (oops!). Unfortunately, the work done here is miles away from being the definitive way towards differentiating the members of IZ*One as it tends to miss the mark *just a bit* sometimes (See Figure 1)

Nevertheless, this remained a fun and interesting project that kept me busy for the better part of 2 months. This writeup aims to be a record of the project. We will describe the journey, methodologies and may go into too many gruesome details that may be uninteresting to the experienced practitioner. For those, I recommend skipping straight to Section A for a detour I took on choosing uniformly random triplets using segment trees before revisiting the discussion sections to judge me on my poor methodology.

And now, for those that remain, we present an awfully formal introduction to this project: IZ*Net is a 2-network system consisting of a face detection model and face recognition model. The system performs the following operations sequentially:

[1] `https://en.wikipedia.org/wiki/Iz*One`

Figure 1: Various labellings by IZ*Net. The top row consists of correct labellings and the bottom labellings are incorrect.[2]

1. Detect and extract faces from the input image using the face detection model
2. Label each individual face using the face recognition model
3. Produce an output image where each extracted face is labelled with a red box and name.

(See Figure 2 for a visual representation of the operations)

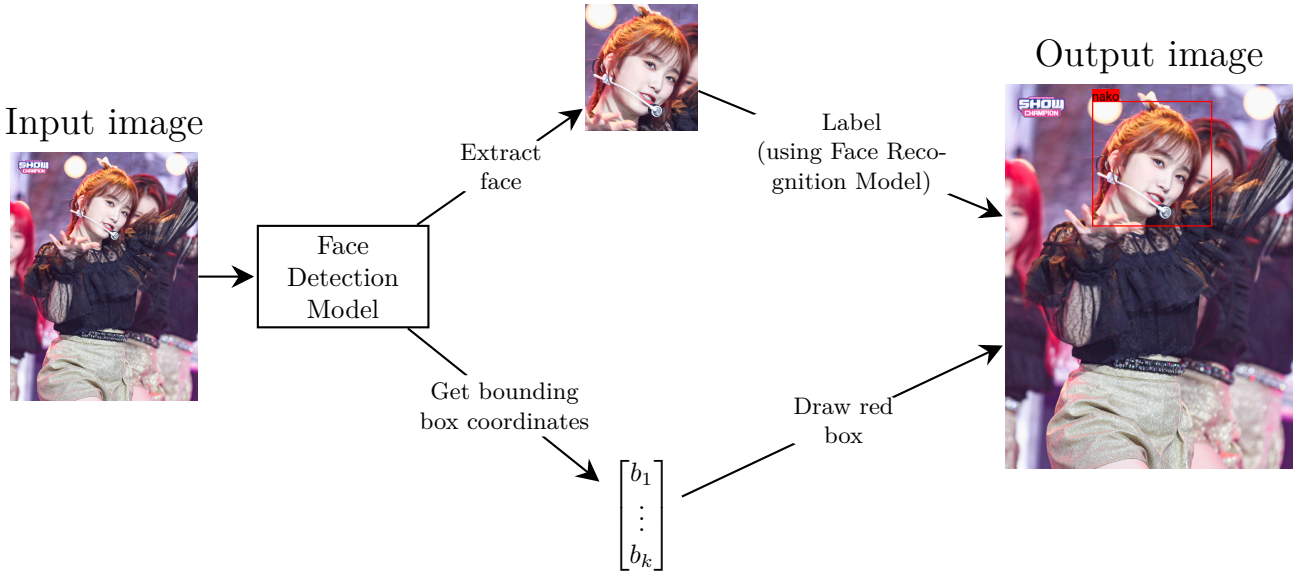


Figure 2: IZ*Net Pipeline

The face detection model is based on the YOLO model [1, 2, 3], while for the face recognition model we experiment with techniques described in the DeepFace [4] and FaceNet [5] models. We present the work on the face recognition model first, in Section 2 since the initial efforts were spent on that. The work on the face detection model is presented in Section 3 and we build the full IZ*Net system in Section 4.

We will include links to code where applicable, for example one can access the Github repo by clicking `this`[3].

2 Face Recognition

2.1 Methodology

The main goal in creating the face recognition model was to experiment over the techniques described in [4, 5]. Specifically, the techniques we will be playing around with are:

1. Minimizing the **triplet loss function** [5, Eq. 1] (`TripletPickerHelper.py`): We use SGD with momentum 0.9 as suggested in the paper, a batch size of 32, and since there around $\binom{n}{3} \in \mathcal{O}(n^3)$ triplets in an n-sized training set, we train (very unscientifically) until it takes too long to obtain a batch of semi-hard triplets. (The real reason for this was because the

[2]Indeed, the face recognition model has trouble with differentiating the JoYuriz trio

[3]I realize this assumes you're on a suitable digital reader, for those that are reading a printed version of this, the repo can be found here: `https://github.com/nicholaspun/IZ-Net`

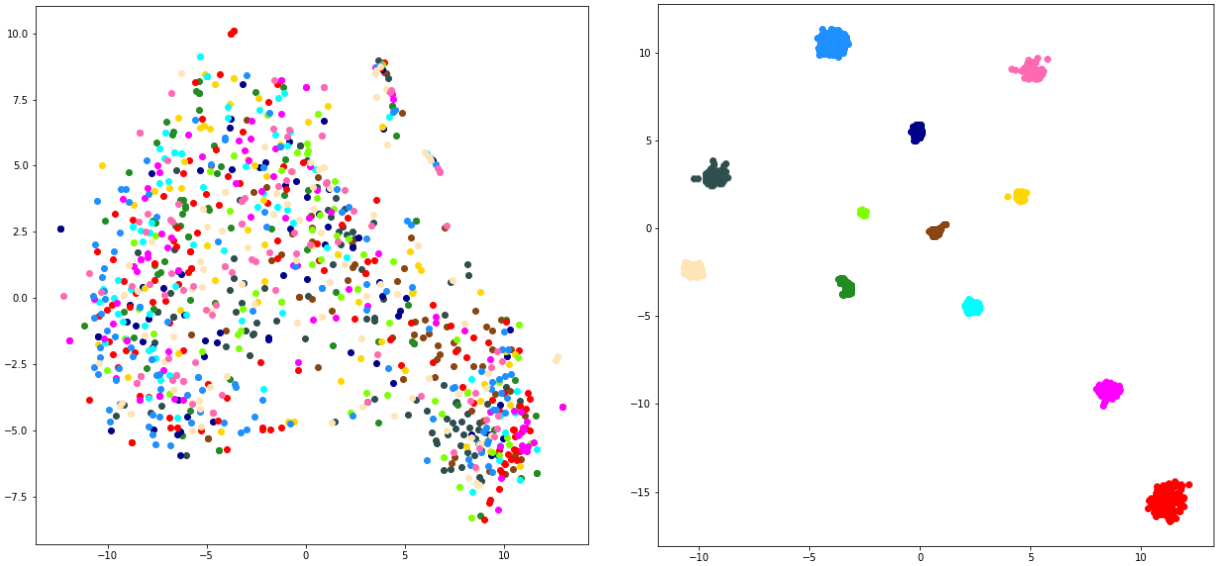
Google Collab notebook would become unstable if left on for too long) This achieved good enough results according to the tools we'll go over in a bit.

2. Casting this as a **binary classification** problem [4, Sec. 4.2] (`PairPickerHelper.py`): Again, we use SGD with momentum 0.9 and a batch size of 32. For binary classification, since choosing and labelling pairs of images is completely deterministic, we *could* train on all $\mathcal{O}(n^2)$ samples. However, we opted to train only on half the amount of possible samples, again, in the interest of time. We note that this many samples was more than sufficient to push the accuracy of the model to be very close to 1 on average.

Both techniques aim to train networks to produce "*suitable*" image embeddings (feature vectors). Here, we define suitable to (roughly) mean that the Euclidean distance between embeddings of images from the same members should be minimized, while embeddings of images from different members should be maximized. To that end, we train 3 different architectures: the "NN1" architecture [5, Table 1], "NN2" architecture [5, Table 2] and Keras InceptionNetV3 architecture[4].

In order to verify our networks, we use 3 different analytical tools (See `analysis.py`)

1. **t-distributed Stochastic Neighbor Embedding** (t-SNE) [6]: We use the *scikit-learn* implementation[5] of t-SNE. It works fairly well out-of-the-box—the only parameters we modified were the number of iterations (increasing it to 5000) and the perplexity (set to 150).[6] These particular values produced sensible results over the range of values we tried out. We use t-SNE to quickly determine the degree of clustering of our training set. Figure 3 shows an example of possible outputs of the algorithm and our aim to is produce an plot that looks like Figure 3b.



(a) Unclustered data

(b) (Probably) clustered data

Figure 3: Possible outputs of the t-SNE algorithm

[4]See `https://keras.io/api/applications/inceptionv3/`

[5]See `https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html`

[6]I particularly enjoyed reading *How to Use t-SNE Effectively* by Wattenberg, Viégas and Johnson as it was quite informative on the consequences of the various parameters

2. **Boxplot of Pairwise Distances**: We compute the pairwise (Euclidean) distances of the image embeddings within the training set for each member and allow *matplotlib* to take care of making the boxplot. We use this tool to quickly determine the dispersion of image embeddings for each member. Ideally we want low dispersion, so we want to see short whiskers and few outliers. This tool is also used in conjunction with the next tool.

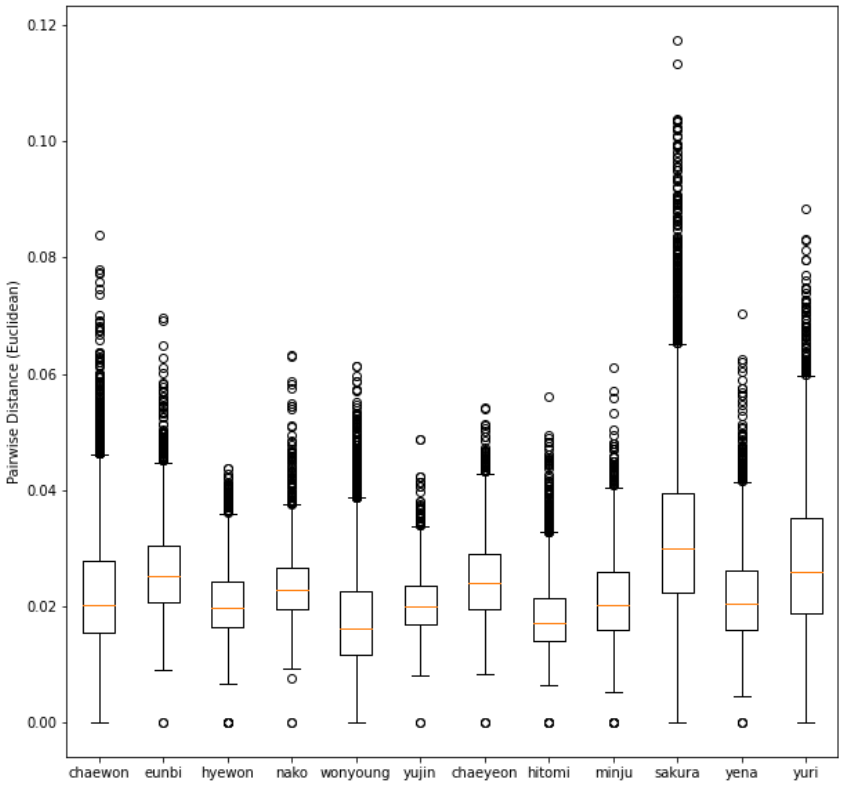


Figure 4: An example boxplot of pairwise distances

3. **Table of medoid distances**: While the boxplot takes care of *intra*-member distances, we also need a tool to help us analyze *inter*-member distances. For each member, we compute the medoid of the image embeddings of their training set and compute the pairwise distances between the medoid for each member. This produces a table like Table 1[7].

   To use this along with the previous tool, we note that the distances between the medoids in Table 1 are at least 1.0. However, in Figure 4, the distances are all less than 0.12. This is an indication that our image embeddings are indeed clustered and there are large distance between clusters.

Finally, we'll take a brief detour to mention how we obtained our training sets. While scraping for images is easy, our training set needs to include images of *only* the face, as in the 2nd step of Figure 2. This is a bit more difficult to come across. To solve this, we cheat a bit and use a pretrained model along with OpenCV to extract faces out of our images—generating a new dataset consisting only of faces.[8]

[7] If you're reading this printed, all the tables are in Section B

[8] Weights were obtained from the Github link located in the article *Extracting faces using OpenCV Face Detection Neural Network* by Bhanot.

2.2 Results and Discussion

While there *should* be 6 different experiments to cover (combinations of 3 different architectures and 2 different training techniques), we accidentally skipped the combination of *InceptionNetV3 and triplet loss training* and didn't notice until the very end. As such, that experiment will not be covered. Otherwise, we will try to go through each experiment systematically. Overall, every experiment achieved fairly decent results.

1. *NN1 and Triplet Loss Training*

   At first glance, the clusters in Figure 5 show that the data is fairly clustered with only a couple stray image embeddings. However, since t-SNE is a probabilistic technique, we must use the other two tools to verify our claim. In particular, we focus on the distances for *Eunbi*in both the boxplot (Figure 6) and medoid table (Table 2). The maximum dispersion the her image embeddings have a distance of 1.2 while the medoid distances can be as low as 1.06 (*Chaeyeon*) and 0.79 (*Minju*). This indicates that there can be overlaps between the clusters of these three members (or at the very least, pairwise overlaps). We make one of two conclusions here (both could occur together):

   (a) We ended training too early. This is quite possible as we didn't have an accuracy metric to work with for triplet loss (More on this in Section 2.3)

   (b) The network was not complex enough to learn differentiating features between the members. (Part of the reason we're trying out different architectures!)

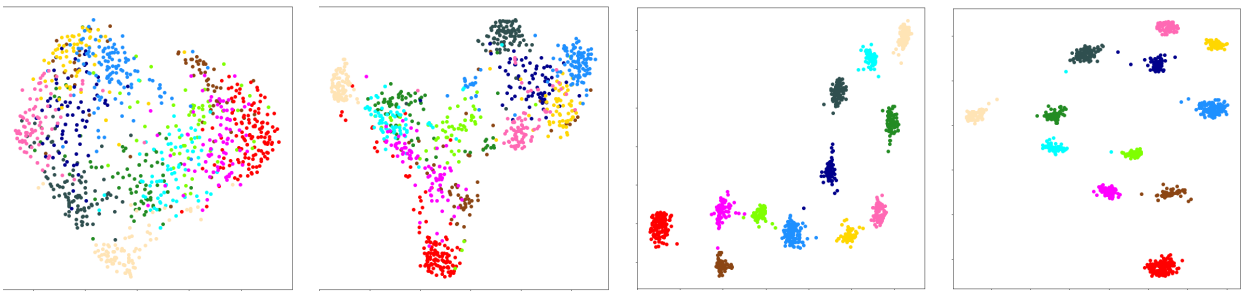


Figure 5: Sequence of t-SNE plots taken at various points during the training of NN1 with the triplet loss technique. From left to right, the plots were taken after training on: 8000 triplets, 16000 triplets, 27520 triplets and 29760 triplets

2. *NN1 and Binary Classification Training*

   The results from this experiment were quite interesting! First, we note that the cluster sizes are quite small in the expanded image of Figure 7. While t-SNE doesn't play well with true cluster sizing, we can confirm (with Figure 8) that, indeed, the dispersion within clusters is quite low. This is exactly the short whiskers and few outliers we were looking for.

   However, on the left hand side of the expanded image, there are two clusters that are overlapping. We can confirm with Table 3 that *Chaewon* and *Yena* have a distance of 0 between their medoids! Further, we note that the pairwise distance between the image embeddings for *Yena* and *Chaewon* (minus a couple outliers) are all 0. This indicates that our network has learned to send all images of *Yena* and *Chaewon* to the same vector[9] in $\mathbb{R}^{128}$.

[9] Insert *yet another* JoYuriz joke here

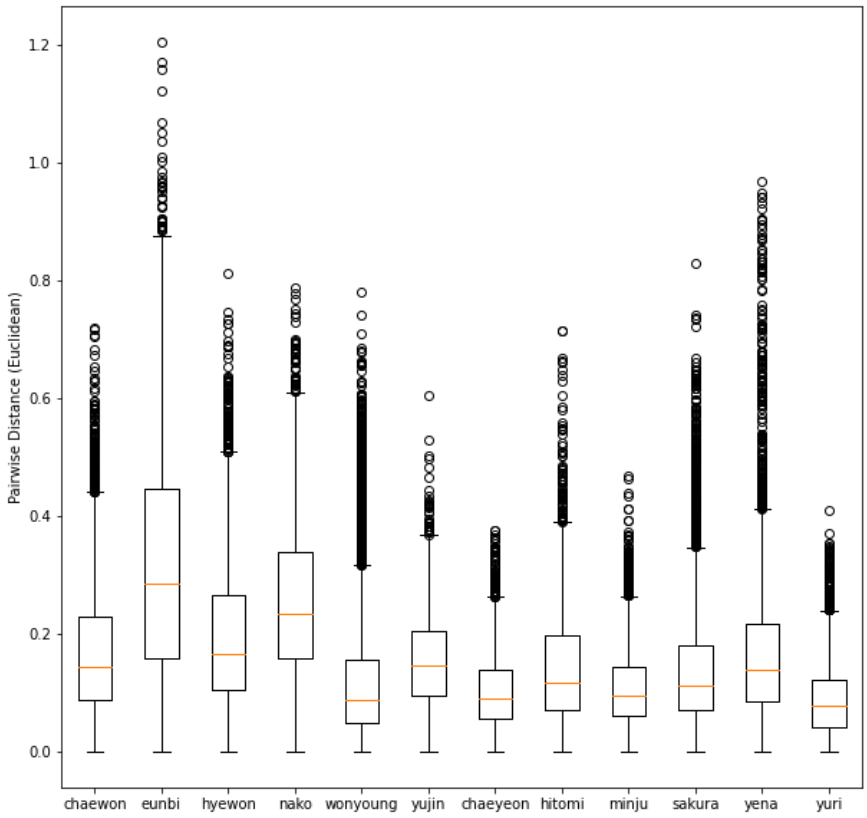


Figure 6: Boxplot of pairwise distances for the training of NN1 with the triplet loss technique

Continuing with the analysis of Table 3, *many* of the distances are exactly 2.0. Since our image embeddings are normalized, this means that our network has learned to send images of each member to their own corner in an 128-simplex. This seems almost too good to be true, and truth be told, this interesting behaviour more likely suggests that something went wrong while training. However, since our implementation of this particular technique worked well with other architectures, we didn't ponder over this too much. Another equally likely possibility is that we completely overfit for our training data.

3. *NN2 and Triplet Loss*

   Recall that NN2 is a much deeper network [5, Table 2], hence we should expect somewhat better results with NN2 than either of our previous experiments. We start by looking at the t-SNE plots (Figure 9). As with our previous experiments, we see what looks to be well-clustered embeddings. Figure 10 tells us that we have intra-cluster distances of up to (around) 0.9 while Table 4 admits inter-cluster distances > 1 for the most part. As such, the three tools tell us that this experiment was fairly successful! The network has learned to minimized distances within clusters and pushed entire clusters far away from each other.

4. *NN2 and Binary Classification Training*

   The results for this experiment is similar to the previous one. The t-SNE plots in Figure 11 show well-defined clusters and we can confirm with Figure 12 that the dispersion of image embeddings within clusters are quite low. Once we examine Table 5, we see that the maximum intra-cluster distance of 0.12 is *very* low compared to the inter-cluster distance, which are all > 1. In fact, solely based on these tools, the best performance came from this experiment!

5. *InceptionNetV3 and Triplet Loss*

   Finally, we get to the last experiment, where we train the pre-built *Keras InceptionV3* with pre-loaded *imagenet* weights. There is not much to say about this experiment since the architecture of InceptionV3 is similar to that of NN2. As such, as one might expect, we get similar looking results as with the *NN2 and Triplet Loss* experiment.

2.3 Closing Remarks

I'll adopt a more casual tone for this subsection. This is here since I wanted one more section to reflect on this portion of the project before moving on. There were a couple things that didn't quite fit into the discussion and analysis in Section 2.2 but that I still wanted to mention.

I'll start off by saying that overall, the training of the face recognition model went well and without too many errors. However, looking back, here's what I would improve:

1. **Develop an accuracy metric**

   While the three tools were useful in determining clustering of the image embeddings, they weren't quite grounded in mathematics. (Or rather, my analysis using the tools were more descriptive than mathematical) The best conclusions I could draw about the trained models were that "it looks like we're doing pretty well" or "there's a couple overlapping clusters here, that's bad!". It would be much easier to compare between models if we looked towards an accuracy metric. A possible metric that I thought of is to look at classification accuracy. (There's some problems with this as the representative embedding for a member could possibly change as the model trains) I'm sure that there are better metrics if I gave this a quick Google search.

Figure 7: Sequence of t-SNE plots taken at various points during the training of NN1 with the binary classification technique. From left-to-right and top-to-bottom, the plots were taken after training on: 0 (Pre-training plot), 500, 1000, 5000, 10000, and 366000 samples. The solo, larger plot is an expanded version of the plot taken at 366000 samples

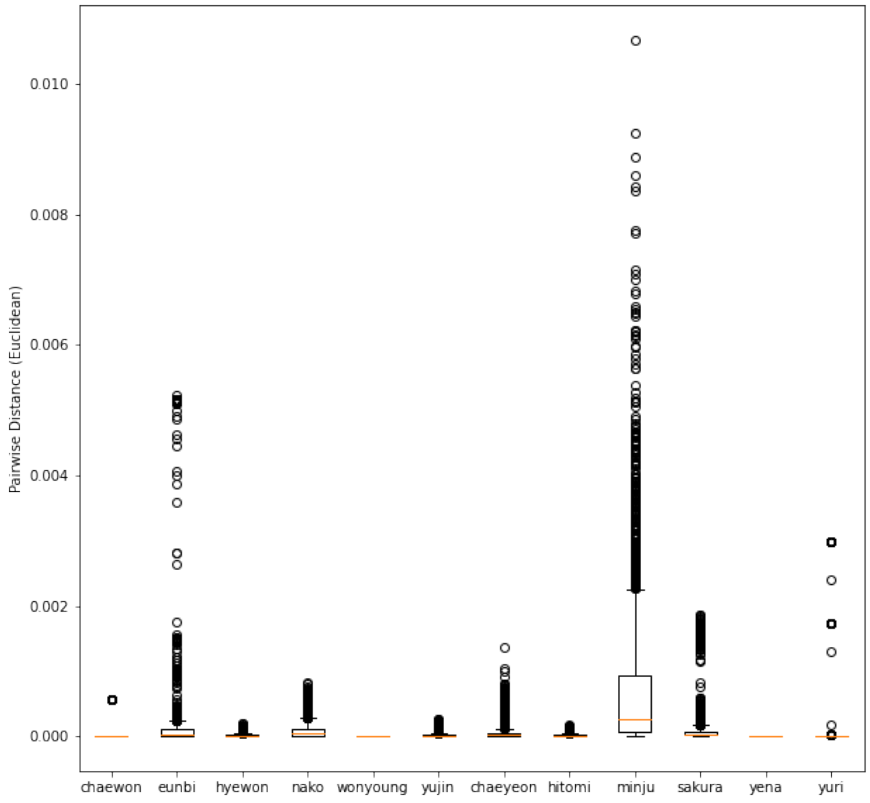


Figure 8: Boxplot of pairwise distances for the training of NN1 with the binary classification technique

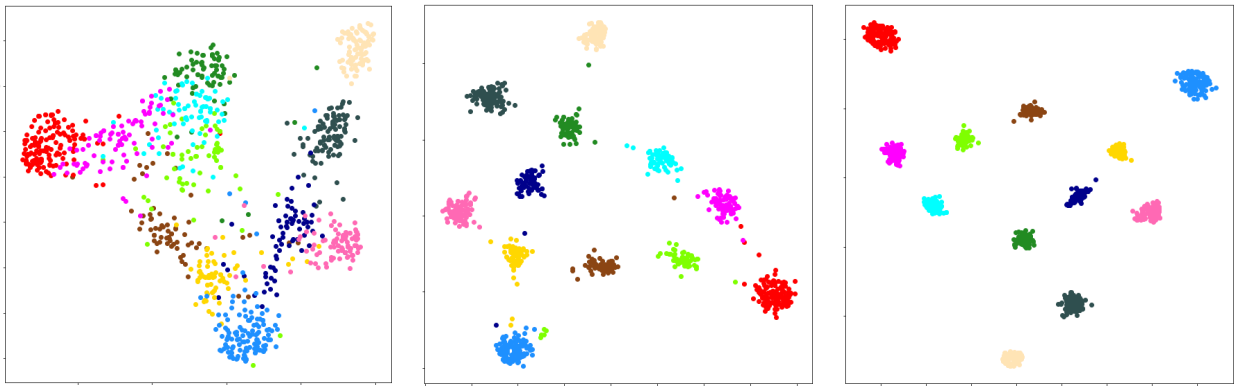


Figure 9: Sequence of t-SNE plots taken at various points during the training of NN2 with the triplet loss function technique. From left-to-right, the plots were taken after training on: 12800, 24320, 24480 samples.

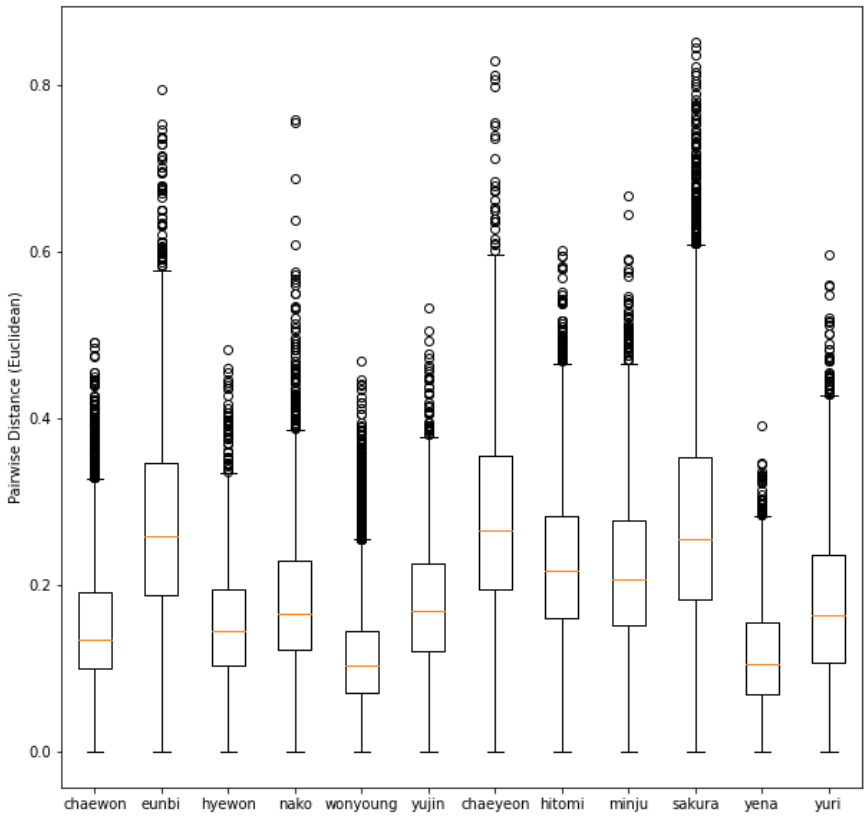


Figure 10: Boxplot of pairwise distances for the training of NN2 with the triplet loss function technique

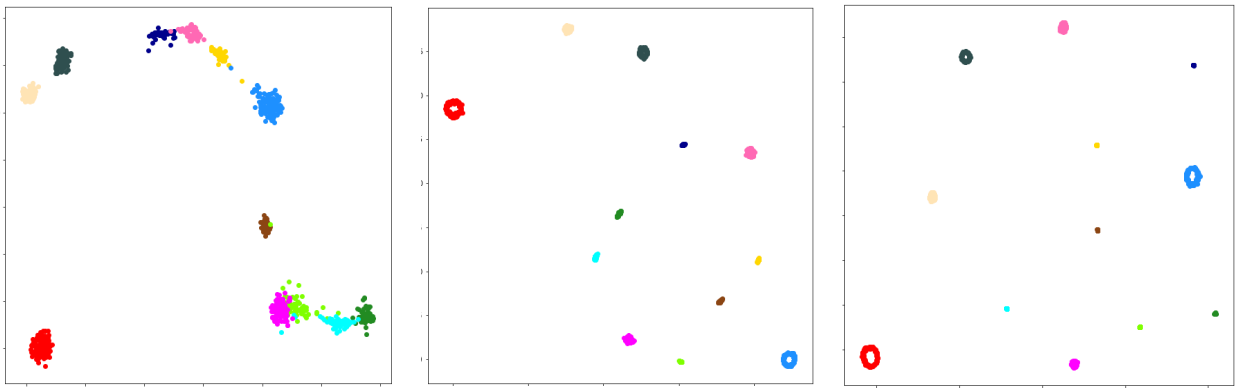


Figure 11: Sequence of t-SNE plots taken at various points during the training of NN2 with the binary classification technique. From left-to-right, the plots were taken after training on: 80000, 160000, 304640 samples.

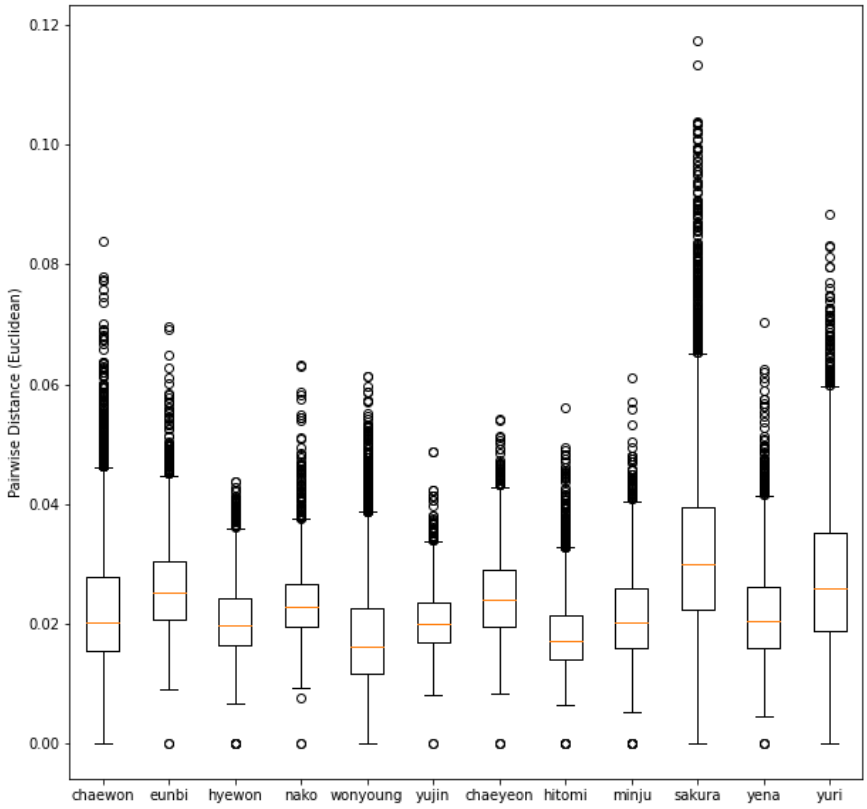


Figure 12: Boxplot of pairwise distances for the training of NN2 with the binary classification technique

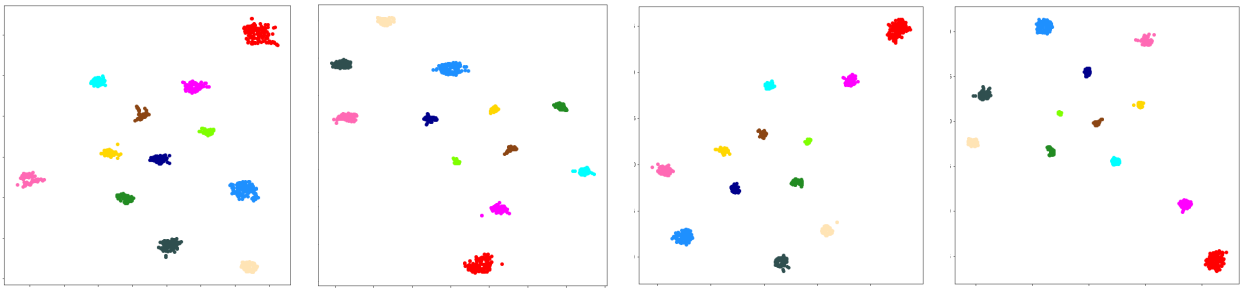


Figure 13: Sequence of t-SNE plots taken at various points during the training of InceptionV3 with the binary classification technique. From left-to-right, the plots were taken after training on: 80000, 160000, 240000, 320000 samples.

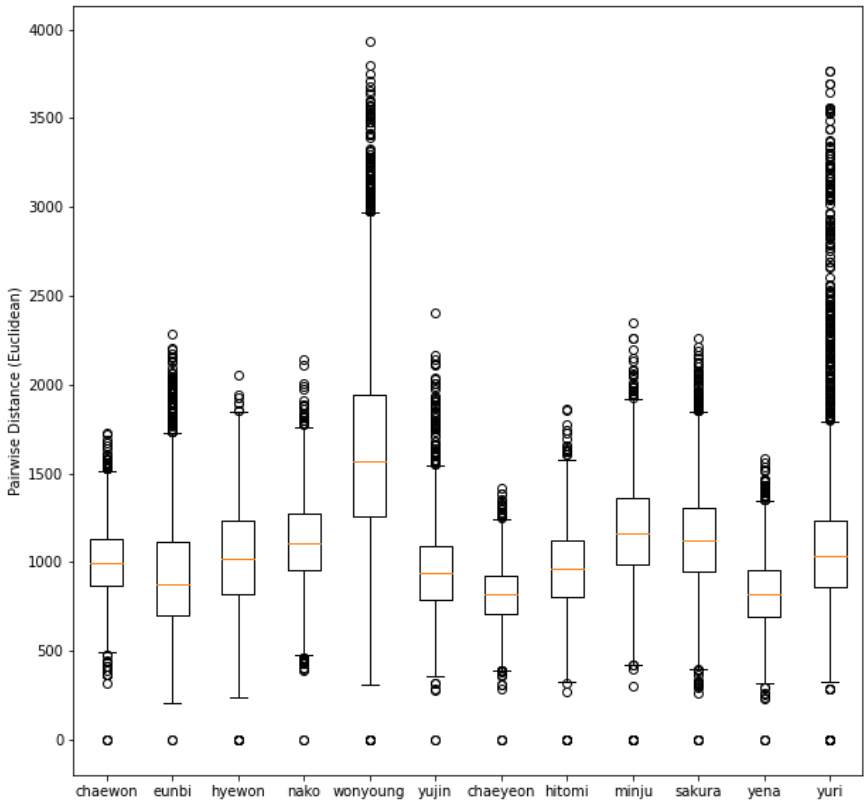


Figure 14: Boxplot of pairwise distances for the training of InceptionV3 Net with the binary classification technique

2. **Split data into training/dev/test sets**

   I actually had the data split as I was scraping for data but merged them near the end since I felt the training set wasn't large enough. I'm fairly sure the models have overfit a bit since in testing the final system we get quite a lot of errors, contrary to what our 3 tools tell us about the accuracy of the model.

   Overall this and the previous remark is just to develop better experimenting procedures that will lead to more controlled experiments.

3. **Fix NN1**

   This is not really an improvement for future projects but something that had me stuck a bit during the creation of NN1. The architecture is not built as specified in [5]. In particular, I had trouble understanding the `fc1` and `fc2` layers, even after reading [7]. To my understanding, the Maxout layer was supposed to be an activation function, but their usage made it slightly confusing and it didn't seem like anyone had implemented this particular architecture on the internet. I ended with doing something completely random that include a `max` function. (Now that I think about this, maybe this is why my 2nd experiment had funky results ...)

3 Face Detection

4 Building the IZ*Net System

A Triplet Mining (with painfully too much tech)

In part of training using triplet loss, we must choose appropriate triplets to feed into our network. There are countless guides and explanations online for how to produce the so-called *semi-hard* and *hard* triplets (over the *easy* ones). But, such is not the focus of this section, instead we are interested in how to choose a uniformly random triplet. Clearly, producing every possible triplet and picking from that set is not very kind to a computer's memory, so the goal is to come up with a scheme that allows to avoid this.

Suppose we have m classes $C_1, \ldots, C_m$, not all necessarily the same size. For convenience, define $\mathscr{C}_i = \bigcup_{\substack{k \neq i \\ k \in \{1,\ldots m\}}} C_k$. Consider the following selection scheme:

Algorithm 1: Uniform Triplet Mining (Take 1)

1 Select $i, j \longleftarrow 1, \ldots m$ uniformly at random (just select one and then the other here). i is the index of the class that will become the anchor
2 Select an random pair of objects in C_i and a random object from C_j

Seems simple enough and also not actually uniformly random. This becomes apparent with a small example:

> Say $C_1 = \{\Gamma, \Lambda\}$ and $C_2 = \{\star, \lozenge, \square\}$. There are a total of 9 different triplets and more importantly the pair (Γ, Λ) participates in 3 of them. However, using Algorithm 1 above, we have a 100% chance of selecting (Γ, Λ) once we choose C_1 as the anchor, which happen 50% of the time. As such, the triplets involving the pair (Γ, Λ) are represented 50% of the time under this scheme, which is not the $\frac{3}{9} \approx 33.33\%$ we calculated above!

There are two issues with Algorithm 1 above:

1. The selection of j is unnecessary: once a pair has been chosen from C_i, the remaining elements in $\mathscr{C}_i$ are all uniformly represented.

   So, we only need to choose i, select a pair from C_i and then select an object uniformly at random from $\mathscr{C}_i$

2. We cannot assign a uniform distribution to the indices $\{1, \ldots, m\}$. Referring back to our example above, note that C_2 produces 3 distinct pairs while C_1 only produces 1. Intuitively, this means we should select C_2 as the anchor more often (in particular, three times as often) so that each pair within the class is equally represented.

   Indeed, let $\mathcal{C} = \binom{C_1}{2} + \ldots \binom{C_m}{2}$. Then, set the probability of choosing C_i as the anchor class to be $\frac{\binom{C_i}{2}}{\mathcal{C}}$.

These two modifications give us the following:

Algorithm 2: Uniform Triplet Mining (Take 2)

1 Select C_i as the anchor class with probability $\frac{\binom{C_i}{2}}{\mathcal{C}}$
2 Select (uniformly at random) a pair of objects from C_i and (again, uniformly at random) an object from $\mathscr{C}_i$

It remains to figure out how to actually perform the first step of Algorithm 2. To do this, we use a data structure known as a Segment Tree [10]. We'll leave the details of the data structure for the reader to explore, but in summary:

- A Segment Tree `T` is created from a set of intervals (over $\mathbb{R}$)
- Given a point `v` $\in \mathbb{R}$, the operation `Find(T, v)` returns a list of intervals from `T` containing `v`

We will find that our particular implementation of the Segment Tree is very specialized: we require the intervals to be disjoint and to complete the real line.

With the preparatory work out of the way, we now describe our strategy for choosing the anchor class:

1. Compute the probabilities for each C_i. For convenience, we'll denote these with p_i so we don't have to repeatedly write out the fraction.
2. Give each C_i an interval from $[0, 1)$. This is easy: give C_1 the interval $[0, p_1)$, give C_2 the interval $[p_1, p_1 + p_2)$, and so on.

   Note that $\sum_{i=1}^{m} p_i = 1$, so C_m is given the interval $[p_1 + \ldots + p_{m-1}, 1)$, completing the interval $[0, 1)$
3. Create a Segment Tree `T` from these intervals. Note that we cheat a bit here and give C_1 the interval $(-\infty, p_1)$ and C_m the interval $[p_1 + \ldots + p_{m-1}, \infty)$. This does not affect the probabilities, it is only an consequence of implementation.

Now, pick a random number between `v` $\in [0, 1]$ (say, using the `random` module in Python). The operation `Find(T, v)` will return exactly one interval, and each interval corresponds to exactly one class. As such, `Find(T, v)` selects classes according to the probability distribution we've defined!

This completes Algorithm 2. We end the section with two remarks:

1. As the section title suggests, is painfully too much tech to choose triplets. For the most part, with large enough datasets and close-to-equal distribution of objects between the classes, these steps can be safely ignored. Not using such tooling will skew the data distribution a tad bit, but the effect on the probabilities will be by very *very* small fractional amounts.
2. We employ a similar strategy for choosing pairs for binary classification, which can be found in `PairPickerHelper.py`. The code for triplet picking can be found in `TripletPickerHelper.py`

[10] See `SegmentTree.py`

B Tables Galore

Unfortunately, I saved all my tables as images and that makes it tough to view in portrait mode. This appendix solves that problem by rotating the page. (I'd actually make this appendix landscape, but that is more LaTeX trouble than I'd like to get into this time around)

	chaewon	eunbi	hyewon	nako	wonyoung	yujin	chaeyeon	hitomi	minju	sakura	yena	yuri
chaewon	0.0	2.9697514	2.2053351	1.8223097	2.465826	2.5563805	1.901767	2.1066985	2.596148	2.1560445	1.6065145	1.6095123
eunbi	2.9697514	0.0	2.620019	2.9911785	1.8885087	1.326247	1.8100963	2.250474	1.4446263	1.751771	2.0387864	2.4853797
hyewon	2.2053354	2.620019	0.0	1.5456016	2.6626284	2.614275	1.5398791	1.5714412	2.3584065	1.9679391	2.5925453	2.0137622
nako	1.8223097	2.9911785	1.5456016	0.0	2.4526138	2.2698116	2.2259529	2.6749902	2.5274692	1.3922213	2.5056639	1.636973
wonyoung	2.465826	1.8885087	2.6626284	2.4526138	0.0	1.9214976	3.021406	2.213265	1.4672607	3.3707006	1.6785339	1.6552181
yujin	2.5563807	1.326247	2.614275	2.2698116	1.9214976	0.0	2.9426043	2.6343575	2.9813182	1.3985581	1.8461761	1.519999
chaeyeon	1.901767	1.8100963	1.5398791	2.2259529	3.021406	2.9426043	0.0	1.866549	1.305156	1.4060404	2.5552123	2.837909
hitomi	2.1066985	2.250474	1.5714413	2.6749902	2.213265	2.6343575	1.8665489	0.0	1.5909499	2.5262575	1.6666744	2.6564515
minju	2.596148	1.4446263	2.3584065	2.5274694	1.4672607	2.9813182	1.305156	1.59095	0.0	2.6316137	2.2271533	2.8266296
sakura	2.1560445	1.751771	1.9679391	1.3922213	3.3707006	1.3985581	1.4060404	2.5262573	2.6316137	0.0	2.6481628	2.4939628
yena	1.6065145	2.0387864	2.5925453	2.505664	1.6785339	1.8461761	2.5552123	1.6666744	2.2271533	2.6481628	0.0	2.5117831
yuri	1.6095123	2.4853797	2.0137625	1.636973	1.6552181	1.519999	2.837909	2.6564515	2.8266296	2.4939628	2.5117831	0.0

Table 1: An example of the table of medoids distances

	chaewon	eunbi	hyewon	nako	wonyoung	yujin	chaeyeon	hitomi	minju	sakura	yena	yuri
chaewon	0.0	1.8235056	1.6133964	0.9352656	1.9631726	1.5283836	1.8576149	1.2574804	1.9463015	1.1996216	1.307389	0.92051893
eunbi	1.8235056	0.0	1.6610799	1.7519772	1.3518933	1.3279886	1.0612085	1.3978806	0.79168296	1.4816688	1.8075213	1.9488937
hyewon	1.6133964	1.6610799	0.0	1.4292123	1.6851559	1.7567859	1.4521444	1.0890496	1.331727	1.9690049	1.3821042	1.3490795
nako	0.9352656	1.7519772	1.4292123	0.0	1.7881818	1.1190042	1.7255241	1.8013588	1.5773115	0.96714157	1.8797266	0.90754604
wonyoung	1.9631726	1.3518933	1.6851559	1.7881818	0.0	1.7707816	1.2421219	1.5586922	1.0775166	1.7949042	1.8466666	1.9713562
yujin	1.5283836	1.3279886	1.7567859	1.1190042	1.7707816	0.0	1.446234	1.728123	1.6232349	0.9066535	1.9786441	0.8243533
chaeyeon	1.8576149	1.0612085	1.4521444	1.7255241	1.2421219	1.446234	0.0	0.903384	0.8313904	0.9320764	1.6055181	1.9784608
hitomi	1.2574804	1.3978806	1.0890496	1.8013588	1.5586922	1.728123	0.903384	0.0	1.0604479	1.7138628	0.97571623	1.9604976
minju	1.9463015	0.79168296	1.331727	1.5773115	1.0775166	1.6232349	0.8313904	1.0604479	0.0	1.9753861	1.7003925	1.9449024
sakura	1.1996216	1.4816688	1.9690049	0.96714157	1.7949042	0.9066535	0.9320764	1.7138629	1.9753861	0.0	1.968653	1.5164721
yena	1.307389	1.8075213	1.3821042	1.8797266	1.8466666	1.9786441	1.6055181	0.97571623	1.7003925	1.968653	0.0	1.8741689
yuri	0.92051893	1.9488937	1.3490795	0.90754604	1.9713562	0.8243533	1.9784608	1.9604976	1.9449024	1.5164721	1.8741689	0.0

Table 2: Medoid distances for the *NN1 and Triplet Loss Training* experiment

	chaewon	eunbi	hyewon	nako	wonyoung	yujin	chaeyeon	hitomi	minju	sakura	yena	yuri
chaewon	0.0	2.0	2.0	1.9999998	2.0	2.0	2.0	1.9999999	2.0	1.0483229	0.0	2.0
eunbi	2.0	0.0	0.8289534	0.9192082	2.0	0.76546013	1.6069272	1.5483078	1.7814732	1.3281851	2.0	2.0
hyewon	2.0	0.8289534	0.0	1.6112306	2.0	1.3430269	0.9605166	0.7215723	1.4749961	1.7621189	2.0	2.0
nako	1.9999998	0.9192082	1.6112306	0.0	1.9999998	1.4685937	1.9999998	1.9999998	1.9999998	0.58350503	1.9999998	1.793715
wonyoung	2.0	2.0	2.0	1.9999998	0.0	2.0	1.1175382	1.9999999	0.2970441	2.0	2.0	2.0
yujin	2.0	0.76546013	1.3430269	1.4685937	2.0	0.0	2.0	1.9999999	2.0	1.6242999	2.0	2.0
chaeyeon	2.0	1.6069272	0.9605166	1.9999998	1.1175382	2.0	0.0	0.21272588	0.31312996	2.0	2.0	2.0
hitomi	1.9999999	1.5483078	0.7215723	1.9999998	1.9999999	1.9999999	0.21272588	0.0	0.9724607	1.9999998	1.9999999	1.9999999
minju	2.0	1.7814732	1.4749961	1.9999998	0.2970441	2.0	0.31312996	0.9724607	0.0	1.9999999	2.0	2.0
sakura	1.0483229	1.3281851	1.7621189	0.58350503	2.0	1.6243	2.0	1.9999998	1.9999999	0.0	1.0483229	0.8206882
yena	0.0	2.0	2.0	1.9999998	2.0	2.0	2.0	1.9999999	2.0	1.0483229	0.0	2.0
yuri	2.0	2.0	2.0	1.7937149	2.0	2.0	2.0	1.9999999	2.0	0.8206882	2.0	0.0

Table 3: Medoid distances for the *NN1 and Binary Classification Training* experiment

	chaewon	eunbi	hyewon	nako	wonyoung	yujin	chaeyeon	hitomi	minju	sakura	yena	yuri
chaewon	0.0	1.6906139	1.1178049	1.2162472	1.7242978	1.9641867	1.1125939	1.2867358	2.1447241	1.5941515	1.0928359	1.2490976
eunbi	1.6906139	0.0	1.6129651	1.4186863	1.0865396	0.92211133	1.082881	1.9352039	1.0479465	1.6954124	1.9162314	1.6859795
hyewon	1.1178049	1.6129651	0.0	0.86289454	1.9865775	1.7593462	1.5845966	1.0169652	1.219909	1.8208263	1.360307	1.1825681
nako	1.2162472	1.4186863	0.86289454	0.0	2.007464	0.8236591	0.92931813	1.4880558	1.6062973	1.3710015	2.151063	0.9312936
wonyoung	1.7242978	1.0865396	1.9865775	2.007464	0.0	1.5760036	1.2276769	2.219986	1.1884121	1.7655964	1.5280545	1.9114013
yujin	1.9641867	0.92211133	1.7593462	0.8236591	1.5760036	0.0	1.604357	2.5101752	1.8712127	1.1262052	2.2310715	0.8364174
chaeyeon	1.1125939	1.082881	1.5845966	0.92931813	1.2276769	1.604357	0.0	1.2658845	1.161425	1.2491097	2.1779256	1.9340398
hitomi	1.2867358	1.9352039	1.0169652	1.4880558	2.219986	2.5101752	1.2658845	0.0	1.208625	1.742432	1.5467033	1.9806349
minju	2.1447241	1.0479465	1.219909	1.6062973	1.1884121	1.8712127	1.161425	1.208625	0.0	2.2210937	1.5703529	2.1101315
sakura	1.5941515	1.6954123	1.8208264	1.3710015	1.7655964	1.1262052	1.2491097	1.742432	2.2210937	0.0	2.3507814	1.7502365
yena	1.0928359	1.9162314	1.360307	2.151063	1.5280545	2.2310717	2.1779256	1.5467033	1.5703528	2.3507814	0.0	1.6495795
yuri	1.2490976	1.6859795	1.1825681	0.9312936	1.9114013	0.8364174	1.9340398	1.9806349	2.1101315	1.7502365	1.6495795	0.0

Table 4: Medoid distances for the *NN2 and Triplet Loss Training* experiment

	chaewon	eunbi	hyewon	nako	wonyoung	yujin	chaeyeon	hitomi	minju	sakura	yena	yuri
chaewon	0.0	2.9697514	2.2053351	1.8223097	2.465826	2.5563805	1.901767	2.1066985	2.596148	2.1560445	1.6065145	1.6095123
eunbi	2.9697514	0.0	2.620019	2.9911785	1.8885087	1.326247	1.8100963	2.250474	1.4446263	1.751771	2.0387864	2.4853797
hyewon	2.2053354	2.620019	0.0	1.5456016	2.6626284	2.614275	1.5398791	1.5714412	2.3584065	1.9679391	2.5925453	2.0137622
nako	1.8223097	2.9911785	1.5456016	0.0	2.4526138	2.2698116	2.2259529	2.6749902	2.5274692	1.3922213	2.5056639	1.636973
wonyoung	2.465826	1.8885087	2.6626284	2.4526138	0.0	1.9214976	3.021406	2.213265	1.4672607	3.3707006	1.6785339	1.6552181
yujin	2.5563807	1.326247	2.614275	2.2698116	1.9214976	0.0	2.9426043	2.6343575	2.9813182	1.3985581	1.8461761	1.519999
chaeyeon	1.901767	1.8100963	1.5398791	2.2259529	3.021406	2.9426043	0.0	1.866549	1.305156	1.4060404	2.5552123	2.837909
hitomi	2.1066985	2.250474	1.5714413	2.6749902	2.213265	2.6343575	1.8665489	0.0	1.5909499	2.5262575	1.6666744	2.6564515
minju	2.596148	1.4446263	2.3584065	2.5274694	1.4672607	2.9813182	1.305156	1.59095	0.0	2.6316137	2.2271533	2.8266296
sakura	2.1560445	1.751771	1.9679391	1.3922213	3.3707006	1.3985581	1.4060404	2.5262573	2.6316137	0.0	2.6481628	2.4939628
yena	1.6065145	2.0387864	2.5925453	2.505664	1.6785339	1.8461761	2.5552123	1.6666744	2.2271533	2.6481628	0.0	2.5117831
yuri	1.6095123	2.4853797	2.0137625	1.636973	1.6552181	1.519999	2.837909	2.6564515	2.8266296	2.4939628	2.5117831	0.0

Table 5: Medoid distances for the *NN2 and Binary Classification Training* experiment

	chaewon	eunbi	hyewon	nako	wonyoung	yujin	chaeyeon	hitomi	minju	sakura	yena	yuri
chaewon	0.0	4482.1333	4291.0166	4820.3394	6324.427	4872.9736	4230.0396	5299.556	5215.7046	4498.4224	3903.6665	5034.061
eunbi	4482.1333	0.0	3590.6753	4475.514	4874.2397	3630.6367	3131.3032	3769.7808	4092.8247	3979.1587	4269.5854	3984.9326
hyewon	4291.0166	3590.6753	0.0	4967.8364	5842.5854	3985.1475	3698.6167	4175.8774	5172.4023	5057.02	4060.4673	4814.7397
nako	4820.3394	4475.514	4967.8364	0.0	7502.7036	4495.469	4335.1064	5204.382	5568.7734	4740.759	4922.1084	5078.1836
wonyoung	6324.427	4874.2397	5842.5854	7502.7036	0.0	7016.5366	5214.2817	5867.731	4445.4727	5824.6426	6348.52	5973.645
yujin	4872.9736	3630.6367	3985.1475	4495.469	7016.5366	0.0	5495.1616	4197.5854	4962.386	4305.02	4740.1304	4451.401
chaeyeon	4230.0396	3131.3032	3698.6167	4335.1064	5214.2817	5495.1616	0.0	3782.5068	4363.1836	4036.4849	4223.656	5531.9214
hitomi	5299.556	3769.7808	4175.8774	5204.382	5867.731	4197.5854	3782.5068	0.0	4811.9707	4737.76	4504.1562	5486.17
minju	5215.7046	4092.8247	5172.4023	5568.7734	4445.4727	4962.386	4363.1836	4811.9707	0.0	5601.4995	5291.495	6228.556
sakura	4498.4224	3979.1587	5057.02	4740.759	5824.6426	4305.02	4036.4849	4737.76	5601.4995	0.0	4244.084	4601.615
yena	3903.6665	4269.5854	4060.4673	4922.1084	6348.52	4740.1304	4223.656	4504.1562	5291.495	4244.084	0.0	5294.3887
yuri	5034.061	3984.9326	4814.7397	5078.1836	5973.645	4451.401	5531.9214	5486.17	6228.556	4601.615	5294.3887	0.0

Table 6: Medoid distances for the *InceptionV3 and Binary Classification Training* experiment

References

[1] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[2] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[3] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[4] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[5] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. 2015.

[6] Geoffrey Hinton and Sam Roweis. Stochastic neighbor embedding. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, page 857–864, Cambridge, MA, USA, 2002. MIT Press.

[7] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. 2013.